

Organización del Computador 2

Recuperatorio del Primer Parcial

27/11/18

Corrige
Mariano

Normas generales

1 (40)	2 (40)	3 (20)	
40	30	20	90 (A)

- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma **no** se incluye en la cantidad total de hojas entregadas.
- Está permitido tener los manuales y los apuntes con las listas de instrucciones en el examen. Está prohibido compartir manuales o apuntes entre alumnos durante el examen.
- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido y LU.
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen tres notas: I (Insuficiente): 0 a 59 pts, A- (Aprobado condicional): 60 a 64 pts y A (Aprobado): 65 a 100 pts. No se puede aprobar con A- ambos parciales. Los recuperatorios tienen dos notas: I: 0 a 64 pts y A: 65 a 100 pts.

Ej. 1. (40 puntos)

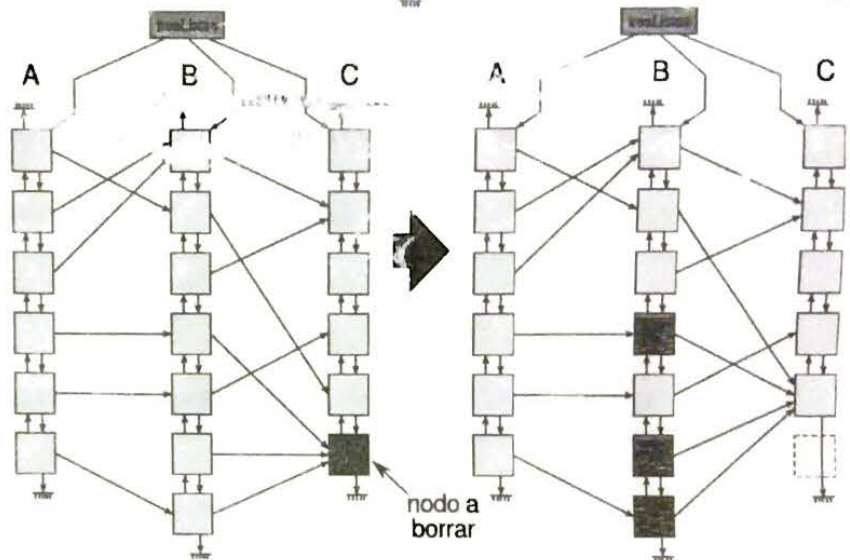
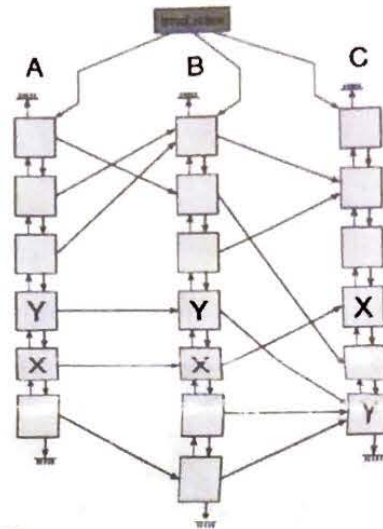
Sea la siguiente estructura del tipo tresListas y su nodo.

```
typedef struct s_tresListas_t {
    nodo* A;
    nodo* B;
    nodo* C;
} tresListas;
```

```
typedef struct s_nodo_t {
    nodo* siguiente;
    nodo* anterior;
    nodo* saltoLista;
    void* dato;
} nodo;
```

tresListas consiste, como su nombre lo indica, en tres listas doblemente enlazadas construidas a partir de los punteros siguiente y anterior de los nodos. Las listas entre sí están ordenadas y pueden tener cualquier cantidad de nodos. Además estas listas están conectadas entre sí por medio del puntero saltoLista. Este último puntero siempre es válido y apunta a un nodo de la lista siguiente en orden.

La figura muestra un ejemplo de las tres listas A, B y C, y como estas están relacionadas. Diremos que un camino es único, si la única forma de llegar a un nodo de la lista C es por medio de un solo nodo en la lista B, que viene de un solo nodo en la lista A. Los nodos señalados con X cumplen esta propiedad, mientras que los nodos Y no la cumplen.



- (20p) a. Implementar en ASM, la función `int esUnico(tresListas* T, nodo* a)`, que toma una estructura de tresListas y un nodo de la primera lista. Recorre los nodos y determina si el camino que comienza por el nodo pasado por parámetro es único. De ser verdadero retornara 1, caso contrario 0.

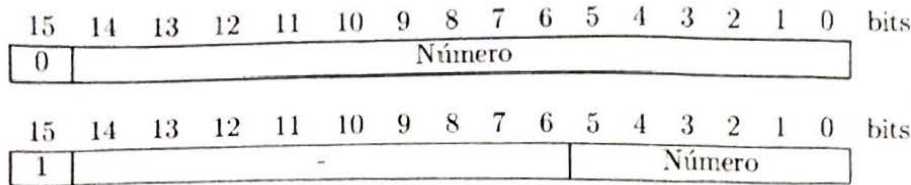
recorro | encuentro → B
busco 1/2
luego que voy a - fin

- (20p) b. Implementar en ASM, la función `void borrarNodoDeC(treslistas* T, nodo* c)`, que toma una estructura de `tresListas` y un nodo de la lista C. Borra este último y modifica todos los punteros que apunten el, haciendolos apuntar al nodo siguiente en la lista. En caso de ser el último, debe apuntar al nodo anterior. Al borrar el nodo no se debe borrar el dato que contiene.

Ej. 2. (40 puntos) *→ poner todos en el mismo estado*

Considerar un tipo de datos de 16 bits denominado **quinceyseis**, que dependiendo del bit más significativo, almacena dos datos diferentes. Si el bit 15 es cero, entonces se almacena un número **con signo** de 15 bits. Si el bit 15 es uno, entonces se almacena un número **sin signo** de 6 bits en la parte menos significativa. Los bits restantes en este caso pueden tener cualquier valor.

0 2⁵⁻¹



*si es 0 → 0
no rep en 6 bit → max*

Sea un vector de **quinceyseis** con tamaño múltiplo de 8. Implementar en ASM utilizando SIMD y procesando la mayor cantidad de datos simultaneamente, las siguientes funciones:

- (20p) a. `void convertirQuince2SeisConSaturacion(quinceyseis* data)`: Transforma todos los números de 15 bits en números de 6 bits saturando.
- (20p) b. `float promedioQuinceSeis(quinceyseis* data)`: Calcula el promedio entre todos los números en *precisión simple*.

Ej. 3. (20 puntos)

Se desea agregar al lenguaje C una funcionalidad especial que permita determinar en tiempo de ejecución que función debe ser ejecutada. Para ello, en el código desarrollado por el usuario se podrá llamar a la función `void generic(size_t indice)`, en donde se determinará qué función es la que se quiere ejecutar, y se procederá a ejecutarla.

Para determinar qué función ejecutar se recibirá en el registro RAX un parámetro **indice** que se utilizará para buscar la función correspondiente en una tabla. Esta tabla se almacenará en la variable global `void (**func_table)(void)`, como doble puntero a función.

Al ejecutar la función buscada en la tabla se debe respetar convención C. Además esta función debe ser ejecutada como si jamás se hubiera llamado a **generic**. Esto significa que el **estado arquitectural** del procesador no debe haber sido alterado de ningún modo posible.

- (4p) a. Implementar en ASM la función **generic**.
- (8p) b. Modificar la función **generic** para que la primera vez que es llamada altere el código de la función que la llamó, reemplazando el llamado a `call` de forma tal que las subsiguientes llamadas se realicen directamente a la función correspondiente.

Nota: Considerar que la instrucción `call` ocupa exactamente 9 bytes, en donde el primer byte es código de operación y los 8 bytes restantes corresponden al puntero a la función.

#C

(a) IDEA

Busco $a \rightarrow b'$

Recorro a Hay Otro En (A, b') a^+

if si $\neq \text{NULL}$

$b \rightarrow c'$

Hay Otro En (B , llegue a c' , distinto de b')

if $\neq \text{NULL}$

return 1

int esUnico ($3 \text{ Listas}^* T, \text{nodo}^* a$)

$\text{nodo}^* a = a$;

$3 \text{ Listas}^* L = T$;

$\text{nodo}^* \text{dest-b} = a \rightarrow \text{saltoLista}$;

$\text{nodo}^* \text{otro} = \text{Hay Otro En} (L \rightarrow A, \text{dest-b}, a)$;

if ($\text{otro} == \text{NULL}$) { // NO hay otro que llege a B.

$\text{nodo}^* \text{dest-c} = b \rightarrow \text{saltoLista}$;

$\text{nodo}^* \text{otro} = \text{Hay Otro En} (L \rightarrow B, \text{dest-c}, \text{dest-b})$;

if ($\text{otro} == \text{NULL}$) { // NO hay otro que llege a C

return 1

} else {

return 0

}

} else {

return 0

}

}

uint32_t #C

bool hoyOtroEn(nodo* listaRecorro, nodo* destino, nodo* yallegoYo){

nodo* actual = listaRecorro;
nodo* otro = NULL;

while(actual != NULL) <sup>cmp = 1
se fin</sup> {
if(actual != yallegoYo) {
nodo* test = actual → salto Lista;
if(test == destino) {
~~// soy yallegoYo pero no llego otro~~
otro = TRUE
}
}
actual = actual → siguiente;
}

return otro; // otro es 1 → hoy otro
// otro es 0 → soy Único :)


```

/ define off salt-sig 0
/ define offset-onterior 8
/ define offset-salto 16
/ define offset-dato 24
/ define NULL 0
/ define lista RBX
" " a R12
" " dest-b R13
" "

```

```

/ define offset-A 0
" " offset-B 8
" " offset-C 16

```

esUnico, RDI T RSI a

```

push rbp
mov rbp, rsp

push lista
push a
push dest-b
sub rsp, 8

mov lista, RDI
mov a, RSI
mov dest-b, QW[a + offset-salto]

mov RDI, QW[lista + offset-A]
mov RSI, dest-b

mov RDX, a

call hoy-otro-en

cmp eax, NULL

jne .fin-habia-otro
; no habia otro

mov RDI, QW[lista + offset-B]
mov RSI, QW[dest-b + offset-salto]
mov RDX, dest-b

call hoy-otro-en
cmp eax, NULL
jne .fin-habia-otro

mov eax, 1, esUnico
jmp .fin

```

```

.fin-habia-otro
xor eax, eax

.fin
add rsp, 8
pop dest-b
pop a
pop lista
pop rbp
ret

```

deja los define abajo

hay-otro-en ; RDI list ESI dest EDX yallego Yo

push rbp

mov rbp, rsp

mov actual, RDI

mov otro, 0 ; soy único

ciclo:

cmp actual, NULL

je .fin

cmp otro, 1

je .fin

cmp actual, EDX

je .sigo

mov test, qw[actual + offset - salto]

cmp test, ESI

jne .sigo

; son iguales y no soy yo ¿ no soy único

inc otro

-sigo:

mov actual, qw[actual + offset - sig]

jmp .ciclo

-fin:

pop rbp

ret

define otro eax

" " actual RS

define test R9

define NULL 0

) esto lo amba

IDEA:

* Si es primero de C \rightarrow actualizo tres listas con next
else

relacionado anterior \rightarrow next y ^{anterior} next.

* Busco next o anterior si este vale NULL y lo paso de parametro

* recorro B y veo quien lo tiene en salto

y lo reemplazo con el nuevo pasado \rightarrow reSet B (nodo^r.3)
x elimino c

void borrarNodoDeC(3l * T, nodo^r c)

nodo^r next \rightarrow nodo^r c = c nodo^r B = T \rightarrow B;

if (c \rightarrow next == NULL)

next = c \rightarrow anterior;

} else {

next = c \rightarrow siguiente;

}

nodo^r lista-C = T \rightarrow C;

if (lista-C == c) // soy primero

T \rightarrow C = next; next \rightarrow anterior = NULL

} else { // mitad de lista

nodo^r anterior = c \rightarrow anterior;

~~c \rightarrow anterior = next~~; anterior \rightarrow next = next

if (next != NULL)

next \rightarrow anterior = anterior;

}

}

sig 2:

~~reSet~~ reSet (B, c, next);

free(c); // no boro el dato (no se que onda)
(quedo malto)

}

```
void reSetB(nod* list, nod* mal_dst, nod* new_dst) {
```

```
    nod* actual = list
```

```
    while(actual->salto != mal_dst)
```

```
    while(actual != NULL) {
```

```
        if(actual->salto == mal_dst) {
```

```
            actual->salto = new_dst;
```

```
        }
```

```
        actual = actual->sig;
```

```
    }
```

```
}
```



```

define B    RBX
" "      C    R12
" "      next R13

```

```

borrarNodoDel: RDI 3e RSI nodo-c

```

```

push rbp
mov rbp, rsp

```

```

push B
push c
push next
sub rsp, 8
mov rcx, RDI
mov c, RSI
mov B, QW[RDI + offset - B]
mov nextR8, QW[c + offset - sig]

```

```

cmp R8, NULL

```

```

jne .siguiente

```

```

mov next, QW[c + offset - anterior]

```

```

jmp .sig0

```

.siguiente:

```

mov next, QW[R + offset - R8]

```

.sig0:

```

mov R8, QW[RDI + offset - c]

```

```

cmp R8, c

```

```

je .sigPrimer0

```

¡mitad lista

```

mov RAX, QW[c + offset - anterior];

```

```

mov QW[RAX + offset - sig], next

```

```

cmp next, NULL

```

```

je .sig2

```

```

.sigPrimer0:
mov QW[RDI + offset - c], next
mov QW[next + offset - anterior], null

```

```

jmp .sig2

```

```

mov QW[next + offset - anterior], RAX

```

.sig2:

```

mov RDI, B

```

```

mov RSI, c

```

```

mov RAX, next

```

```

call reSet

```

```

mov RDI, c

```

```

call free

```

```

add rsp, 8

```

```

pop next

```

```

pop c

```

```

pop B

```

```

pop rbp

```

```

ret

```

```
1 define null 0  
1 define actual RDI  
1 " mal-dst RSI  
1 define new_dst RDX
```

```
resetB: ; RDI lista RSI dst-mal RDX dst-beno
```

```
push rbp
```

```
mov rbp, rsp
```

```
mov
```

```
cmp actual, null
```

```
je -fin
```

```
mov R8, QW[actual + offset - salto]
```

```
cmp R8, mal-dst
```

```
jne -sig
```

```
mov QW[actual + offset - salto], new_dst
```

```
-sig: mov actual, QW[actual + offset - sig]
```

```
jmp -ciclo
```

```
pop rbp
```

```
ret
```


Primero hice el (b). aco' abajo están los defines que valen en (a)

* sector con datos de size word 2 bytes, lee 8 datos x vez

* si es múltiplo de 8, luego es de 4 tmb

* mejor proceso de 0 a 8

* tenemos 2 casos

si bit 15 es 0, ^{no} lo dejo como estaba máscara con

0s en 15 30 45 60 75 90 105 bla no importa

La máscara-select-0 \Rightarrow los modifico

si es 1 los dejo \Rightarrow cmp x_1, x_2 y shifted word left

* caso a modificar

. extendiendo signo a word shift arit. \rightarrow
no son 15 a 6 bits todos 0, no representable
si usa el bit 6, problema mask-15-6-0

Lo debo saturar todo 1

. si último 15 es 1 (neg) \rightarrow todo 0

mask-15-1

Lo pand extendiendo shift ar \rightarrow 1111 000 1111

convertir

/. define x x mm

/. define unos x mm

/. define ceros x 10

/. define mask-15-1 x 15

/. define mask-15-6-1 x 14

si
~~no~~ modificar

~~no~~ modificar
~~no se pudo con los otros~~

convertir 1546:

push rbp
mov rbp, esp

shr esi, 3, mov 8

cmpq b ^{unos} ~~0~~, unos

pxor zeros, zeros

movq mask_15_1, unos

psllw mask_15_1, 31

movq mask_15_6, ~~0~~, unos

psllw mask_15_6_1, 6

ciclo:

cmp esi, 0

~~mov~~ je .fn

movq x0, [rdi]

movq x1, x0

pand x1, mask_15_1, el
elijo los 10 bits fuertes

psraw x1, 31

movq x2, x0

pand x2, x1
x2 ← datos ya son 6

pxor x1, unos

x1 ← mask a modificar

psllw x0, 4

psraw x0, 4

, extendi signo

movq x3, x0

~~cmpq~~ ~~pxor~~ x3, mask_15_4

psllw x3, 31

, si neg 1 else 0000

, caso negativo pongo todos a 0
; de vuelta la máscara elijo los
~~movq x4, x0~~
que voy a cambiar

pxor x3, unos, el negativo

pand x0, x3, ^{lo a ser 0}
; 0 → 0, ^{el and}

~~movq x4, x0~~

~~movq x4, x0~~

, solo viven ^{pos. unos} ~~negativos~~

movq x4, x0

pand x4, mask_15_6_1

cmpeqw x4, zeros

, todos word debiesen tener 0
; ~~no~~ representable con saturar

pxor x4, unos

, elijo los que voy a saturar

~~movq x5, x0~~
~~pand x5, x4~~ ← saturar

pand x0, x4 ; no representable

pand x0, x1

, solo seteo esos en 1111 todo
; en x2 tengo elegidos los que iba a
modificar

~~movq [rdi], x0~~

por x0, x2

, de los datos que estaban

bien

movq [rdi], x0

dec esi

add rdi, 16

jmp .ciclo

pop rbp

ret

(b) Vuelvo a elegir proceso de 4 (me da lo repetido código binario)

La const seis → shift todo a 15

poniendo Quince Seis, RDI data size

push rbp
mov rbp, rsp
mov eax, esi
shr eax, 3, para 8
pxor x0, x0, como parcial
cmpeq b unos, unos

pxor unos, unos
movq mask, 15, 1, unos
psllw mask, 15, 1 (31)

ciclo

cmp eax, 0
je fin
movq x1, [rdi]
movq x2, x1
pand x2, mask, 15
para q x2 (31)
elijo los de 15
movq x3, x1

pand x3, x2, de 6 en x3 (Al Reves)

psllw x3, 26
psraw x3, 26
baje su signo
pxor x2, unos
; máscara los de 15

pand x1, x2
psllw x1, 1
psraw x1, 1

divido el signo de 15
por x1, x2, merge de ambos casos
pumpexh

movq x2, x1, e-h
pumpexhwd x1, unos, low
pumpexhwd x2, unos, high
cvttsd2ps x1, x1
cvttsd2ps x2, x2
addps x1, x2
haddps x0, x1

dec esi
add rdi, 16, al final proceso 8
jmp ciclo

; en x0 tengo suma parcial
haddps x0, unos, 11111111
haddps x0, unos, 11111111

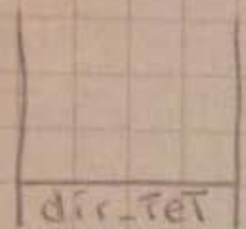
1-1-1-1-1-1-1-1

pinstr xmm1, eax, 0, inserto m
cvttsd2ss xmm1, xmm1
; convert to scalar into scalar float
; ~~no es~~ single precision

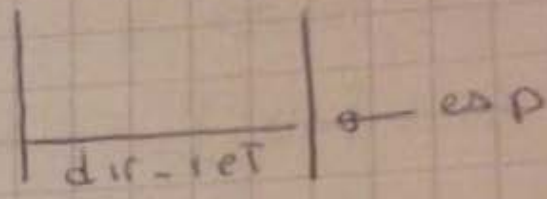
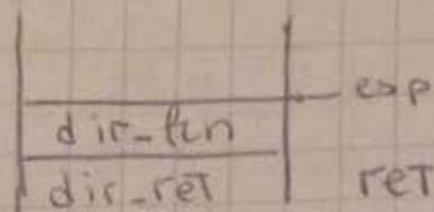
divss x0, x1
pop rbp
ret

(a)

Notemos que si hago call, se pusha lo dir de retorno donde pertenece a la función que hace call. Debo usar RET ejecuta fun



call functionGeneric



como si nada hubiera pasado (al menos en la pila).

generic: ; RAX índice
mov RAX, [RSP - 16]
; no puedo modificar registros

shl RAX, 3 ; add fun-table, 8
RAX * 8

~~add fun-table, RAX;~~
~~mov RAX, [fun-table]~~

me gustó A

opción A add RAX, fun-table

mov RAX, [RAX]

⊗

push RAX

mov RAX, [RSP - 8]

Ret

(b) Resuelto atrás

me gustó A

opción A

add rax, fun-table

mov rax, [rax]

ⓧ

push rax

mov rax, [rsp-8]

ret

(b) Resuelto atrás

Para alterar el código, tengo lo dir-ret en la pila. Retrocedo 9 bytes y puedo escribir en el call generic, pero me piden que altere código a futuro. Puedo leer esos 9 bytes y guardarlos.

En realidad solo necesito 1 byte del CALL pues tengo etiqueta del "generic". ~~Entonces~~ Busco desde lo dir-ret de la pila en adelante buscando coincidencias. ¿Hasta cuánto?

Natalia

Consulté y me dijeron que solo modifique esta vez que me llamaron, pues no se cuánto duro el código. ~~y~~

push rsp

generic:

```
mov [rsp-24], RAX
```

en RAX tengo la dir de la función

```
(ADD RAX, fun-table  
mov RAX, [RAX])
```

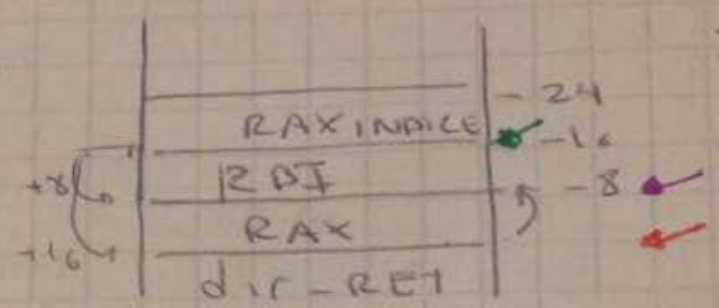
push RAX ; así cuando hago RET yo onda

push RDI ; veamos si lo puedo usar preservando

```
mov RDI, [rsp+16]
```

```
sub RDI, 8
```

```
mov [RDI], RAX
```



; RDI ← dir-RET

escribo call genérico función

```
pop RDI      pop RDI
```

```
mov RDI, [rsp+16]      mov RAX, [rsp-16] ; RAX ← indice
```

```
RET
```